

# C (langage)

C est un langage de programmation impératif et généraliste. Inventé au début des années 1970 pour réécrire UNIX, C est devenu un des langages les plus utilisés. De nombreux langages plus modernes comme C++, C#, Java et PHP reprennent des aspects de C.



Date de première version	1972
Paradigme	Impératif, procédural, structuré
Auteur	Dennis Ritchie
Développeur	Dennis Ritchie, Bell Labs
Dernière version	C11
Typage	Statique, faible
Normes	ANSI X3.159-1989 (ANSI C, C89) ISO/CEI 9899:1990 (C90) ISO/CEI 9899:1999 (C99) ISO/CEI 9899:2011 (C11)
Influencé par	BCPL, B, Algol 68, Fortran
A influencé	awk, csh, C++, C#, Objective-C, D, Concurrent C, Java, JavaScript, PHP, Perl
Implémentations	GCC, MSVC, Borland C,

# Sommaire

---

## Histoire

- Normalisation

## Caractéristiques générales

- Qualités et défauts
- Aperçu de la syntaxe
  - Hello world*
- Évolution des pratiques
- Brièveté de la syntaxe
- Langage d'expressions

## Des sources à l'exécutable

- Sources
- Précompilation
- Compilation
- Assemblage
- Édition des liens

## Éléments du langage

- Éléments lexicaux
- Mots clés
- Instructions du préprocesseur
- Types
- Structures
- Commentaire
- Structures de contrôle
- Fonctions
  - Prototype
- Comportements ambigus
  - Comportements définis par l'implémentation
  - Comportements non spécifiés
  - Comportements indéfinis
- Exemples

## Bibliothèques logicielles

- La bibliothèque standard
- Les bibliothèques externes

## Exemples

- Allocation mémoire

## Références

### Voir aussi

- Articles connexes
  - Quelques programmes célèbres écrits en C
- Bibliographie

Clang, TCC

### Extensions de fichiers

.c, .h

# Histoire

---

Le langage C a été inventé au cours de l'année 1972 dans les Laboratoires Bell. Il était développé en même temps que UNIX par Dennis Ritchie et Ken Thompson. Ken Thompson avait développé un prédécesseur de C, le langage B, qui est lui-même inspiré de BCPL. Dennis Ritchie a fait évoluer le langage B dans une nouvelle version suffisamment différente, en ajoutant notamment les types, pour qu'elle soit appelée C<sup>1</sup>.

Bien que C soit officiellement inspiré de B et de BCPL, on note une forte influence de PL/I (ou de PL360) ; on a pu dire que C était à Unix et au PDP-11 ce que PL/I fut pour la réécriture de Multics.

Par la suite, Brian Kernighan aida à populariser le langage C. Il procéda aussi à quelques modifications de dernière minute.

En 1978, Kernighan fut le principal auteur du livre *The C Programming Language* décrivant le langage enfin stabilisé ; Ritchie s'était occupé des appendices et des exemples avec Unix. On appelle aussi ce livre « le K&R », et l'on parle de **C traditionnel** ou de **C K&R** lorsqu'on se réfère au langage tel qu'il existait à cette époque.



Ken Thompson (à gauche) et Dennis Ritchie (à droite).

## Normalisation

En 1983, l'Institut national américain de normalisation (ANSI) a formé un comité de normalisation (X3J11) du langage qui a abouti en 1989 à la norme dite **ANSI C** ou **C89** (formellement ANSI X3.159-1989). En 1990, cette norme a également été adoptée par l'Organisation internationale de normalisation (**C90**, **C ISO**, formellement ISO/CEI 9899:1990). ANSI C est une évolution du C K&R qui reste extrêmement compatible. Elle reprend quelques idées de C++, notamment la notion de prototype et les qualificateurs de type<sup>2</sup>.

Entre 1994 et 1996, le groupe de travail de l'ISO (ISO/CEI JTC1/SC22/WG14) a publié deux correctifs et un amendement à C90 : ISO/CEI 9899/COR1:1994 *Technical Corrigendum 1*, ISO/CEI 9899/AMD1:1995 *Intégrité de C* et ISO/CEI 9899/COR1:1996 *Technical Corrigendum 2*. Ces changements assez modestes sont parfois appelés C89 avec amendement 1, ou C94 / C95<sup>3,4</sup>. Trois fichiers d'entêtes ont été ajoutés, dont deux concernant les caractères larges et un autre définissant un certain nombre de macros en rapport avec la norme de caractères ISO 646.

En 1999, une nouvelle évolution du langage est normalisée par l'ISO : **C99** (formellement ISO/CEI 9899:1999). Les nouveautés portent notamment sur les tableaux de taille variable, les pointeurs restreints, les nombres complexes, les littéraux composés, les déclarations mélangées avec les instructions, les fonctions *inline*, le support avancé des nombres flottants, et la syntaxe de commentaire de C++. La bibliothèque standard du **C99** a été enrichie de six fichiers d'en-tête depuis la précédente norme.

En 2011, l'ISO ratifie une nouvelle version du standard<sup>5</sup> : **C11**, formellement ISO/CEI 9899:2011. Cette évolution introduit notamment le support de la programmation multi-thread, les expressions à type générique, et un meilleur support d'Unicode.

## Caractéristiques générales

C est un langage de programmation impératif et généraliste. Il est qualifié de langage de bas niveau dans le sens où chaque instruction du langage est conçue pour être compilée en un nombre d'instructions machine assez prévisible en termes d'occupation mémoire et de charge de calcul. En outre, il propose un éventail de types entiers et flottants conçus pour pouvoir correspondre directement aux types de donnée supportés par le processeur. Enfin, il fait un usage intensif des calculs d'adresse mémoire avec la notion de pointeur<sup>6</sup>.

Hormis les types de base, C supporte les types énumérés, composés, et opaques. Il ne propose en revanche aucune opération qui traite directement des objets de plus haut niveau (fichier informatique, chaîne de caractères, liste, table de hachage...). Ces types plus évolués doivent être traités en manipulant des pointeurs et des types composés. De même, le langage ne propose pas en standard la gestion de la programmation orientée objet, ni de système de gestion d'exceptions. Il existe des fonctions standards pour gérer les entrées-sorties et les chaînes de caractères, mais contrairement à d'autres langages, aucun opérateur spécifique pour améliorer l'ergonomie. Ceci rend aisé le remplacement des fonctions standards par des fonctions spécifiquement conçues pour un programme donné.

Ces caractéristiques en font un langage privilégié quand on cherche à maîtriser les ressources matérielles utilisées, le langage machine et les données binaires générées par les compilateurs étant relativement prévisibles. Ce langage est donc extrêmement utilisé dans des domaines comme la programmation embarquée sur microcontrôleurs, les calculs intensifs, l'écriture de systèmes d'exploitation et les modules où la rapidité de traitement est importante. Il constitue une bonne alternative au langage d'assemblage dans ces domaines, avec les avantages d'une syntaxe plus expressive et de la portabilité du code source. Le langage C a été inventé pour écrire le système d'exploitation UNIX, et reste utilisé pour la programmation système. Ainsi le noyau de grands systèmes d'exploitation comme Windows et Linux sont développés en grande partie en C.

En contrepartie, la mise au point de programmes en C, surtout s'ils utilisent des structures de données complexes, est plus difficile qu'avec des langages de plus haut niveau. En effet, dans un souci de performance, le langage C impose à l'utilisateur de programmer certains traitements (libération de la mémoire, vérification de la validité des indices sur les tableaux...) qui sont pris en charge automatiquement dans les langages de haut niveau.

Dépouillé des commodités apportées par sa bibliothèque standard, C est un langage simple, et son compilateur l'est également. Cela se ressent au niveau du temps de développement d'un compilateur C pour une nouvelle architecture de processeur : Kernighan et Ritchie estimaient qu'il pouvait être développé en deux mois car « on s'apercevra que les 80 % du code d'un nouveau compilateur sont identiques à ceux des codes des autres compilateurs existant déjà. »<sup>7</sup>

## Qualités et défauts

C'est un des langages les plus utilisés car :

- il existe depuis longtemps, le début des années 1970 ;
- il est fondé sur un standard ouvert ;
- de nombreux informaticiens le connaissent ;
- étant du plus bas niveau portable possible, il permet la minimisation de l'allocation mémoire nécessaire et la maximisation de la performance, notamment par l'utilisation de pointeurs ;
- des compilateurs et bibliothèques logicielles existent sur la plupart des architectures ;
- il a influencé de nombreux langages plus récents donc C++, Java, C# et PHP ; sa syntaxe en particulier est largement reprise ;
- il met en œuvre un nombre restreint de concepts, ce qui facilite sa maîtrise et l'écriture de compilateurs simples et rapides ;
- il ne spécifie pas rigide le comportement du fichier exécutable produit, ce qui aide à tirer parti des capacités propres à chaque ordinateur ;
- il permet l'écriture de logiciels qui n'ont besoin d'aucun support à l'exécution (la bibliothèque logicielle ni machine virtuelle), au comportement prévisible en temps d'exécution comme en consommation de mémoire vive, comme des noyaux de système d'exploitation et des logiciels embarqués

Ses principaux inconvénients sont :

- le peu de vérifications offertes par les compilateurs d'origine (K&R C), et l'absence de vérifications à l'exécution, ce qui fait que des erreurs qui auraient pu être automatiquement détectées lors du développement ne l'étaient qu'à l'exécution, donc au prix d'un plantage du logiciel ;
  - sous UNIX, on pouvait utiliser les utilitaires lint et cflow pour éviter de tels mécomptes ;
  - des vérifications sont ajoutées avec le temps, mais elles restent partielles ;
- son approche de la modularité restée au niveau de ce qui se faisait au début des années 1970, et largement dépassée depuis par d'autres langages :
  - il ne facilite pas la programmation orientée objet ;
  - il ne permet pas de créer des espaces de noms ;
- la gestion d'exception très sommaire ;
- le support très limité de la généricité, malgré l'introduction des expressions à type générique en C11 ;
- les subtilités de l'écriture de programmes portables, car le comportement exact des exécutables dépend de l'ordinateur cible ;
- le support minimaliste de l'allocation de mémoire et des chaînes de caractères ce qui oblige les programmeurs à s'occuper de détails fastidieux et sources d'bugs ; il n'y a notamment pas de ramasse-miettes standard ;

- les bugs graves qui peuvent être causés par un simple manque d'attention du développeur ; tel le dépassement de tampon qui constitue une faille de sécurité informatique exploitable par les logiciels malveillants ;
- certaines erreurs ne peuvent être détectées automatiquement qu'à l'aide d'outils supplémentaires et non standardisés, comme lint et Valgrind ;
- la faible productivité du langage par rapport aux langages plus récents <sup>[réf. souhaitée]</sup>.

## Aperçu de la syntaxe

### *Hello world*

Le programme *Hello world* est proposé en exemple en 1978 dans *The C Programming Language* de Brian Kernighan et Dennis Ritchie. Créer un programme affichant *hello world* est depuis devenu l'exemple de référence pour présenter les bases d'un nouveau langage. Voici l'exemple original de la 1<sup>re</sup> édition de 1978 :

```
main()
{
    printf("hello, world\n");
}
```

- `main` est le nom de la fonction principale, aussi appelée point d'entrée du programme.
- Les parenthèses ( ) après `main` indiquent qu'il s'agit d'une fonction.
- Les accolades { et } entourent les instructions constituant le corps de la fonction.
- `printf` est une fonction d'écriture dans la sortie standard, qui produit l'affichage dans la console par défaut.
- Les caractères " délimitent une chaîne de caractères; `hello, world\n` dans ce cas.
- Les caractères `\n` sont une séquence d'échappement représentant le saut de ligne.
- Un point-virgule ; termine l'instruction expression.

### Évolution des pratiques

Le même programme, conforme à la norme ISO et suivant les bonnes pratiques contemporaines :

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

- `#include <stdio.h>` inclut l'en-tête standard `<stdio.h>` contenant les déclarations des fonctions d'entrées-sorties de la bibliothèque standard du C, dont la fonction `printf` utilisée ici.
- `int` est le type renvoyé par la fonction `main`. Le type `int` est le type implicite en K&R C et en C89, et il était couramment omis du temps où l'exemple de Kernighan et Ritchie a été écrit. Il est obligatoire en C99.
- Le mot-clé `void` entre parenthèses signifie que la fonction n'a aucun paramètre. Il peut être omis sans ambiguïté lors de la définition d'une fonction. En revanche, s'il est omis lors de la déclaration de la fonction, cela signifie que la fonction peut recevoir n'importe quels paramètres. Cette particularité de la déclaration est considérée comme obsolète dans le standard en C 2011<sup>10</sup>. On peut noter que dans le standard MISRA C 2004, qui impose des restrictions au C89 pour des usages nécessitant une plus grande sûreté, le mot-clé `void` est obligatoire à la déclaration comme à la définition d'une fonction sans arguments<sup>10</sup>.
- L'instruction `return 0;` indique que la fonction `main` retourne la valeur 0. Cette valeur est de type `int`, et correspond au `int` devant le `main`.

### Brièveté de la syntaxe

La syntaxe de C a été conçue pour être brève. Historiquement, elle a souvent été comparée à celle de Pascal, langage impératif également créé dans les années 1970. Voici un exemple avec une fonction factorielle :

```
/* En C (norme ISO) */
int factorielle(int n) {
    return n > 1 ? n * factorielle(n - 1) : 1;
}
```

```
{ En Pascal }
function factorielle(n: integer) : integer
begin
    if n > 1 then factorielle := n * factorielle(n - 1)
    else factorielle := 1
end.
```

Là où Pascal utilise des mots clés `function`, `integer`, `begin`, `if`, `then`, `else` et `end`, C n'utilise que `int` et `return`, les autres mots clés étant remplacés par des parenthèses et accolades, ainsi que l'opérateur `?:`.

## Langage d'expressions

La brièveté de C ne repose pas que sur la syntaxe. Le grand nombre d'opérateurs disponibles, le fait que la plupart des instructions contiennent une expression, que les expressions produisent presque toujours une valeur et que les instructions de test se contentent de comparer la valeur de l'expression testée avec zéro, concourent à la brièveté du code source.

Voici l'exemple de fonction de copie de chaîne de caractères — dont le principe est de copier les caractères jusqu'à avoir copié le caractère nul, qui marque par convention la fin d'une chaîne en C — donné dans *The C Programming Language*, 2<sup>e</sup> édition, p. 106 :

```
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

La boucle `while` utilise un style d'écriture classique en C, qui a contribué à lui donner une réputation de langage peu lisible. L'expression `*s++ = *t++` contient : deux déréférencements de pointeur ; deux incrémentations de pointeur ; une affectation ; et la valeur affectée est comparée avec zéro par le `while`. Cette boucle n'a pas de corps, car toutes les opérations sont effectuées dans l'expression de test du `while`. On considère qu'il faut maîtriser ce genre de notation pour maîtriser <sup>11</sup>C.

Pour comparaison, une version n'utilisant pas les opérateurs raccourcis ni la comparaison implicite à zéro donnerait :

```
void strcpy(char *s, char *t)
{
    while (*t != '\0') {
        *s = *t;
        s = s + 1;
        t = t + 1;
    }
    *s = *t;
}
```

## Des sources à l'exécutable

### Sources

Un programme écrit en C est généralement réparti en plusieurs fichiers sources compilés séparément.

Les fichiers sources C sont des fichiers texte, généralement dans le codage des caractères du système hôte. Ils peuvent être écrits avec un simple éditeur de texte. Il existe de nombreux éditeurs, voire des environnements de développement intégrés (IDE), qui ont des fonctions spécifiques pour supporter l'écriture de sources en C.

L'usage est de donner les extensions de nom de fichier .c et .h aux fichiers source C. Les fichiers .h sont appelés *fichiers d'en-tête*, de l'anglais *header*. Ils sont conçus pour être inclus au début des fichiers source, et contiennent uniquement des déclarations.

Lorsqu'un fichier .c ou .h utilise un identificateur déclaré dans un autre fichier .h, alors il inclut ce dernier. Le principe généralement appliqué consiste à écrire un fichier .h pour chaque fichier .c, et à déclarer dans le fichier .h tout ce qui est exporté par le fichier .c.

La génération d'un exécutable à partir des fichiers sources se fait en plusieurs étapes, qui sont souvent automatisées à l'aide d'outils comme make, SCons, ou bien des outils spécifiques à un environnement de développement intégré. Les étapes menant des sources au fichier exécutable sont au nombre de quatre : précompilation, compilation, assemblage, édition de liens. Lorsqu'un projet est compilé, seuls les fichiers.c font partie de la liste des fichiers à compiler ; les fichiers.h sont inclus par les directives du préprocesseur contenues dans les fichiers source.

## Précompilation

Le préprocesseur C exécute des directives contenues dans les fichiers sources. Il les reconnaît au fait qu'elles sont en début de ligne, et commencent toutes avec le caractère croisillon #. Parmi les directives les plus courantes, il y a :

- `#include` pour l'inclusion ;
- `#define` pour la définition d'un macro ;
- `#if` pour commencer la compilation conditionnelle ;
- `#ifdef` et `#ifndef`, équivalents à `#if defined` et `#if ! defined` ;
- `#endif` pour clore la compilation conditionnelle.

Outre l'exécution des directives, le préprocesseur remplace les commentaires par un espace blanc, et procède au remplacement des macros. Pour le reste, le code source est transmis tel quel au compilateur pour la phase suivante. Il faut toutefois que chaque `#include` dans le code source soit récursivement remplacé par le code source inclus. Ainsi, le compilateur reçoit un seul source du préprocesseur, qui constitue l'unité de compilation.

Voici un exemple de fichier source `copyarray.h` faisant un usage classique des directives du préprocesseur :

```
#ifndef COPYARRAY_H
#define COPYARRAY_H

#include <stddef.h>

void copyArray(int *, size_t);

#endif
```

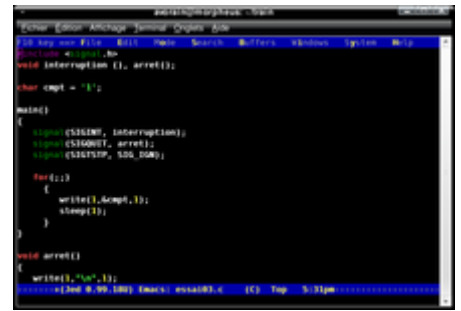
Les directives `#ifndef`, `#define` et `#endif` garantissent que le code à l'intérieur n'est compilé qu'une seule fois même s'il est inclus plusieurs fois. La directive `#include <stddef.h>` inclut l'en-tête qui déclare le type `size_t` utilisé plus bas.

## Compilation

La phase de compilation consiste généralement en la génération du code assembleur. C'est la phase la plus intensive en traitements. Elle est accomplie par le compilateur proprement dit. Pour chaque unité de compilation, on obtient un fichier en langage d'assemblage.

Cette étape peut être divisée en sous-étapes :

- l'analyse lexicale, qui est la reconnaissance des mots clé du langage ;



Édition d'un fichier source C avec l'éditeur de texte Jed.

- l'analyse syntaxique qui analyse la structure du programme et sa conformité avec la norme ;
- l'optimisation de code ;
- l'écriture d'un code isomorphe à celui de l'assembleur (et parfois du code assembleur lui-même quand cela est demandé en option du compilateur).

Par abus de langage, on appelle compilation toute la phase de génération d'un fichier exécutable à partir des fichiers sources. Mais c'est seulement une des étapes menant à la création d'un exécutable.

Certains compilateurs C fonctionnent à ce niveau en deux phases, la première générant un fichier compilé dans un langage intermédiaire destiné à une machine virtuelle idéale (voir P-Code) portable d'une plate-forme à l'autre, la seconde convertissant le langage intermédiaire en langage d'assemblage dépendant de la plate-forme cible. D'autres compilateurs C permettent de ne pas générer de langage d'assemblage, mais seulement le fichier compilé en langage intermédiaire, qui sera interprété ou compilé automatiquement en code natif à l'exécution sur la machine cible (par un machine virtuelle qui sera liée au programme final).

## Assemblage

---

Cette étape consiste en la génération d'un fichier objet en langage machine pour chaque fichier de code assembleur. Les fichiers objet sont généralement d'extension `.o` sur Unix, et `.obj` avec les outils de développement pour MS-DOS, Microsoft Windows, VMS, CP/M... Cette phase est parfois regroupée avec la précédente par établissement d'un flux de données interne sans passer par des fichiers en langage intermédiaire ou langage d'assemblage. Dans ce cas, le compilateur génère directement un fichier objet.

Pour les compilateurs qui génèrent du code intermédiaire, cette phase d'assemblage peut aussi être totalement supprimée : c'est une machine virtuelle qui interprétera ou compilera ce langage en code machine natif. La machine virtuelle peut être un composant du système d'exploitation ou une bibliothèque partagée.

## Édition des liens

---

L'édition des liens est la dernière étape, et a pour but de réunir tous les éléments d'un programme. Les différents fichiers objet sont alors réunis, ainsi que les bibliothèques statiques, pour ne produire qu'un fichier exécutable.

Le but de l'édition de liens est de sélectionner les éléments de code utiles présents dans un ensemble de codes compilés et de bibliothèques, et de résoudre les références mutuelles entre ces différents éléments afin de permettre à ceux-ci de se référencer directement à l'exécution du programme. L'édition des liens échoue si des éléments de code référencés manquent.

# Éléments du langage

---

## Éléments lexicaux

Le jeu de caractères ASCII suffit pour écrire en C. Il est même possible, mais inusité, de se restreindre au jeu de caractères invariants de la norme ISO 646, en utilisant des séquences d'échappement appelées trigraphe. En général, les sources C sont écrits avec le jeu de caractères du système hôte. Il est toutefois possible que le jeu de caractères d'exécution ne soit pas celui du source.

Le C est sensible à la casse. Les caractères blancs (espace, tabulation, fin de ligne) peuvent être librement utilisés pour la mise en page, car ils sont équivalents à une seule espace dans la plupart des cas.

## Mots clés

Le C89 compte 32 mots clés, dont cinq qui n'existaient pas en K&R C, et qui sont par ordre alphabétique :

auto, break, case, char, const (C89), continue, default, do, double, else, enum (C89), extern, float, for, goto, if, int, long, register, return, short, signed (C89), sizeof, static, struct, switch, typedef, union, unsigned, void (C89), volatile (C89), while.



Ce sont des termes réservés qui ne doivent pas être utilisés autrement.

La révision C99<sup>12</sup> en ajoute cinq :

`_Bool`, `_Complex`, `_Imaginary`, `inline`, `restrict`.

Ces nouveaux mots-clés commencent par une majuscule préfixée d'un *underscore* afin de maximiser la compatibilité avec les codes existants. Des en-têtes de la bibliothèque standard fournissent les alias `bool` (`<stdbool.h>`), `complex` et `imaginary` (`<complex.h>`).

La dernière révision, C11<sup>13</sup>, introduit encore sept nouveaux mots-clés avec les mêmes conventions :

`_Alignas`, `_Alignof`, `_Atomic`, `_Generic`, `_Noreturn`, `_Static_assert`, `_Thread_local`.

Les en-têtes standards `<stdalign.h>`, `<stdnoreturn.h>`, `<assert.h>` et `<threads.h>` fournissent respectivement les alias `alignas` et `alignof`, `noreturn`, `static_assert`, et `thread_local`.

## Instructions du préprocesseur

---

Le préprocesseur du langage C offre les directives suivantes :

`#include`, `#define`, `#pragma` (C89), `#if`, `#ifdef`, `#ifndef`, `#elif` (C89), `#else`, `#endif`, `#undef`, `#line`, `#error`.

## Types

Le langage C comprend de nombreux types de nombres entiers, occupant plus ou moins de bits. La taille des types n'est que partiellement standardisée : le standard fixe uniquement une taille minimale et une magnitude minimale. Les magnitudes minimales sont compatibles avec d'autres représentations binaires que le complément à deux bien que cette représentation soit presque toujours utilisée en pratique. Cette souplesse permet au langage d'être efficacement adapté à des processeurs très variés, mais elle complique la portabilité des programmes écrits en C.

Chaque type entier a une forme « signée » pouvant représenter des nombres négatifs et positifs, et une forme « non signée » ne pouvant représenter que des nombres naturels. Les formes signées et non signées doivent avoir la même taille.

Le type le plus commun est `int`, il représente le mot machine.

Contrairement à de nombreux autres langages, le type `char` est un type entier comme un autre, bien qu'il soit généralement utilisé pour représenter les caractères. Sa taille est par définition d'un byte.

## Types entiers, en ordre croissant

Type	Taille	Magnitude <i>minimale</i> exigée par le standard <sup>14</sup>   Bits
<u>char</u>	-127 à 127, ou 0 à 255, selon l'implémentation	≥ 8 bits
signed char	-127 à 127	
unsigned char (C89)	0 à 255	
<u>short</u> signed short	-32 767 à +32 767	≥ 16 bits
unsigned short	0 à 65 535	
<u>int</u> signed int	-32 767 à +32 767	≥ 16 bits
unsigned int	0 à 65 535	
<u>long</u> signed long	-2 147 483 647 à +2 147 483 647	≥ 32 bits
unsigned long	0 à 4 294 967 295	
long long (C99) signed long long (C99)	-9 223 372 036 854 775 807 à +9 223 372 036 854 775 807	≥ 64 bits
unsigned long long (C99)	0 à 18 446 744 073 709 551 615	

Les types énumérés se définissent avec le mot clé `enum`.

Il existe des types de nombre à virgule flottante, de précision, donc de longueur en bits, variable ; en ordre croissant :

## Types décimaux, en ordre croissant

Type	Précision	Magnitude
float	≥ 6 chiffres décimaux	environ $10^{-37}$ à $10^{+37}$
double	≥ 10 chiffres décimaux	environ $10^{-37}$ à $10^{+37}$
long double	≥ 10 chiffres décimaux	environ $10^{-37}$ à $10^{+37}$
long double (C89)	≥ 10 chiffres décimaux	

C99 a ajouté `float complex`, `double complex` et `long double complex`, représentant les nombres complexes associés.

Types élaborés :

- `struct`, `union`, `*` pour les pointeurs ;
- `[... ]` pour les tableaux ;
- `(... )` pour les fonctions.

Le type `_Bool` est standardisé par C99. Dans les versions antérieures du langage, il était courant de définir un synonyme :

```
typedef enum boolean {false, true} bool;
```

Le type `void` représente le vide, comme une liste de paramètres de fonction vide, ou une fonction ne retournant rien.

Le type `void*` est le pointeur générique : tout pointeur de donnée peut être implicitement converti de et vers `void*`. C'est par exemple le type retourné par la fonction standard `malloc`, qui alloue de la mémoire. Ce type ne se prête pas aux opérations nécessitant de connaître la taille du type pointé (arithmétique de pointeurs, déréférencement).

## Structures

C supporte les types composés avec la notion de *structure*. Pour définir une structure, il faut utiliser le mot-clé `struct` suivit du nom de la structure. Les membres doivent ensuite être déclarés entre accolades. Comme toute déclaration, un point-virgule termine le tout.

```
/* Déclaration de la structure personne */
struct Personne
{
    int age;
    char *nom;
};
```

Pour accéder aux membre d'une structure, il faut utiliser l'opérateur `.`.

```
int main()
{
    struct Personne p;
    p.nom = "Albert";
    p.age = 46;
}
```

Les fonctions peuvent recevoir des pointeurs vers des structures. Ils fonctionnent avec la même syntaxe que les pointeurs classiques. Néanmoins, l'opérateur `->` doit être utilisé sur le pointeur pour accéder aux champs de la structure. Il est également possible de déréférencer le pointeur pour ne pas utiliser cet opérateur et toujours utiliser l'opérateur `.`.

```
void anniversaire(struct Personne * p)
{
    p->age++;
    printf("Joyeux anniversaire %s !" , (*p).nom);
}

int main()
{
    struct Personne p;
    p.nom = "Albert";
    p.age = 46;
    anniversaire(&p);
}
```

## Commentaire

Dans les versions de C antérieures à C99, les commentaires devaient commencer par une barre oblique et un astérisque (« `/*` ») et se terminer par un astérisque et une barre oblique. Tout ce qui est compris entre ces symboles est du commentaire, saut de ligne compris :

```
/* Ceci est un commentaire
sur deux lignes
ou plus */
```

La norme C99 a repris de C++ les commentaires de fin de ligne, introduits par deux barres obliques et se terminant avec la ligne :

```
// Commentaire jusqu'à la fin de la ligne
```

## Structures de contrôle

La syntaxe des différentes structures de contrôle existantes en C est largement reprise dans plusieurs autres langages, comme le C++ bien sûr, mais également Java, C#, PHP ou encore JavaScript.

Les trois grands types de structures sont présents :

- les tests (également appelés branchements conditionnels) avec :
  - `if (expression) instruction`
  - `if (expression) instruction else instruction`
  - `switch (expression) instruction`, avec `case` et `default` dans l'instruction
- les boucles avec :
  - `while (expression) instruction`
  - `for (expression_optionnelle ; expression_optionnelle ; expression_optionnelle) instruction`
  - `do instruction while (expression)`
- les sauts (branchements inconditionnels) :
  - `break`
  - `continue`
  - `return expression_optionnelle`
  - `goto étiquette`

## Fonctions

Les fonctions en C sont des blocs d'instructions, recevant un ou plusieurs arguments et pouvant retourner une valeur. Si une fonction ne retourne aucune valeur, le mot-clé `void` est utilisé. Une fonction peut également ne recevoir aucun argument. Le mot-clé `void` est conseillé dans ce cas.

```
// Fonction ne retournant aucune valeur (appelée procédure)
void afficher(int a)
{
    printf("%d", a);
}

// Fonction retournant un entier
int somme(int a, int b)
{
    return a + b;
}

// Fonction sans aucun argument
int saisir(void)
{
    int a;
    scanf("%d", &a);
    return a;
}
```

## Prototype

Un prototype consiste à déclarer une fonction et ses paramètres sans les instructions qui la composent. Un prototype se termine par un point-virgule.

```
// Prototype de saisir
int saisir(void);

// Fonction utilisant saisir
int somme(void)
{
    int a = saisir(), b = saisir();
    return a + b;
}
```

```

}
// Définition de saisir
int saisir(void)
{
    int a;
    scanf("%d", &a);
    return a;
}

```

Généralement, tous les prototypes sont écrits dans des fichiers `h`, et les fonctions sont définies dans un fichier `c`.

## Comportements ambigus

La norme du langage C laisse, délibérément, certaines opérations sans spécification précise. Cette propriété du C permet aux compilateurs d'utiliser directement des instructions spécifiques au processeur, d'effectuer des optimisations ou d'ignorer certaines opérations, pour compiler des programmes exécutables courts et efficaces. En contrepartie, c'est parfois la cause de bugs de portabilité des codes source écrits en C.

Il existe trois catégories de tels comportements<sup>15</sup> :

- **défini par l'implémentation**: Le comportement n'est pas spécifié dans la norme mais dépend de l'implémentation. Le choix effectué dans une implémentation doit être documenté dans celle-ci. Un programme utilisant ce type de comportement est correct, à défaut d'être garanti portable.
- **non spécifié**: Le choix n'est pas spécifié dans la norme, mais n'a cette fois pas à être documenté. Il n'a en particulier pas à être identique à chaque fois pour une même implémentation. Un programme utilisant ce type de comportement est correct également.
- **non défini**: Comme le nom l'indique, l'opération n'est pas définie. La norme n'impose aucune limitation à ce que le compilateur peut faire dans ce cas. Tout peut arriver. Le programme est incorrect.

## Comportements définis par l'implémentation

En C, les comportements définis par l'implémentation<sup>16</sup> sont ceux où l'implémentation doit choisir un comportement et s'y tenir. Ce choix peut être libre ou parmi une liste de possibilités données par la norme. Le choix doit être documenté par l'implémentation, afin que le programmeur puisse le connaître et l'utiliser.

Un des exemples les plus importants de tel comportement est la taille des types de donnée entiers. La norme C spécifie la taille minimale des types de base, mais pas leur taille exacte. Ainsi, le type `int` par exemple, correspondant au mot machine, doit avoir une taille minimale de 16 bits. Il peut avoir une taille de 16 bits sur un processeur 16 bits et une taille de 64 bits sur un processeur 64 bits.

Un autre exemple est la représentation des entiers signés<sup>17</sup>. Il peut s'agir du complément à deux, du complément à un ou d'un système avec un bit de signe et des bits de valeur (en). La vaste majorité des systèmes modernes utilise le complément à deux, qui est par exemple le seul encore supporté par GCC<sup>18</sup>. De vieux systèmes utilisent les autres formats, comme l'IBM 7090 qui utilise le format signe/valeur, le PDP-1 ou l'UNIVAC et ses descendants, dont certains encore utilisés actuellement tels le UNIVAC 1100/2200 series#UNISYS 2200 series(en), qui utilisent le complément à un.

Un autre exemple est le décalage à droite d'un entier signé négatif<sup>19</sup>. Typiquement, l'implémentation peut choisir de décaler comme pour un entier non signé ou de propager le bit de poids fort représentant le signe.

## Comportements non spécifiés

Les comportements non spécifiés<sup>20</sup> sont similaires aux comportements définis par l'implémentation, mais le comportement adopté par l'implémentation n'a pas à être documenté. Il n'a même pas à être le même en toute circonstance. Néanmoins, le programme reste correct, le programmeur ne peut juste pas compter sur une règle particulière.

Par exemple, l'ordre d'évaluation des paramètres lors d'un appel de fonction n'est pas spécifié. Le compilateur peut même choisir d'évaluer dans un ordre différents les paramètres de deux appels à la même fonction, si ça peut aider son optimisation.

## Comportements indéfinis

La norme C définit certains cas où des constructions syntaxiquement valides ont un comportement indéfini<sup>21</sup>. Selon la norme, tout peut alors arriver : la compilation peut échouer, ou produire un exécutable dont l'exécution sera interrompue, ou qui produira des résultats faux, ou même qui donnera l'apparence de fonctionner sans erreur. Lorsqu'un programme contient un comportement indéfini, c'est le comportement de l'ensemble du programme qui devient indéfini, pas seulement le comportement de l'instruction contenant l'erreur. Ainsi, une instruction erronée peut corrompre des données qui seront traitées bien plus tard, reportant d'autant la manifestation de l'erreur. Et même sans être exécutée, une instruction erronée peut amener le compilateur à réaliser des optimisations sur la base d'hypothèses fausses, produisant un exécutable qui ne fait pas du tout ce qui est prévu.

## Exemples

On peut signaler la classique division par zéro, ou l'affectation multiple d'une variable dans la même expression avec l'exemple<sup>22</sup> :

```
int i = 4;
i = i++; /* Comportement indéfini. */
```

On pourrait ainsi penser que dans cet exemple `i` pourrait valoir 4 ou 5 suivant le choix du compilateur, mais il pourrait tout aussi bien valoir 42 ou l'affectation pourrait arrêter l'exécution, ou le compilateur peut refuser la compilation. Aucune garantie n'existe dès qu'un comportement indéfini existe.

Pour ne citer que quelques exemples, le déréférencement d'un pointeur nul, tout accès à un tableau hors de ses limites<sup>23</sup>, l'utilisation d'une variable non initialisée ou encore le débordement d'entiers signés ont tous des comportements indéfinis. Le compilateur peut utiliser le fait qu'une construction est indéfinie dans certains cas pour supposer que ce cas ne se produit jamais et optimiser plus agressivement le code. Si l'exemple ci-dessus peut paraître évident, certains exemples complexes peuvent être bien plus subtils et être source de bugs parfois graves<sup>24,25</sup>.

Par exemple, beaucoup de code contient des vérifications destinées à éviter l'exécution dans des cas hors bornes, qui peut ressembler à ceci<sup>26</sup> :

```
char buffer[BUFLen];
char *buffer_end = buffer + BUFLen;
unsigned int len;

/* ... */

if (buffer + len >= buffer_end || /* vérification de dépassement du buffer */
    buffer + len < buffer)      /* vérification de débordement si len très large */
    return;

/* Si pas de débordement, effectue les opérations prévues */
/* ... */
```

En apparence, ce code est prudent et effectue les vérifications de sécurité nécessaires pour ne pas déborder du buffer alloué. En pratique, les versions récentes de compilateurs tels que GCC, Clang ou Microsoft Visual C++ peuvent supprimer le second test, et rendre possibles des débordements. En effet, la norme précise que l'arithmétique de pointeur sur un objet ne peut donner un pointeur hors de cet objet. Le compilateur peut donc décider que le test est toujours faux et le supprimer. La vérification correcte est la suivante :

```
char buffer[BUFLen];
unsigned int len;

/* ... */

if (len >= BUFLen) /* vérification de dépassement du buffer */
    return;
```

```
/* Si pas de débordement, effectue les opérations prévues */  
/* ... */
```

En 2008, quand les développeurs de GCC ont modifié le compilateur pour qu'il optimise certaines vérifications de débordement qui reposaient sur des comportements indéfinis, le CERT a émis un avertissement sur l'utilisation des versions récentes de GCC<sup>27</sup>. Ces optimisations sont en fait présentes dans la plupart des compilateurs modernes, le CERT a révisé son avertissement dans ce sens.

Certains outils existent pour détecter ces constructions problématiques, et les meilleurs compilateurs en décèlent certaines (il faut parfois activer des options particulières) et peuvent les signaler, mais aucun ne prétend à l'exhaustivité.

## Bibliothèques logicielles

### La bibliothèque standard

La bibliothèque standard normalisée, disponible avec toutes les implémentations, présente la simplicité liée à un langage bas-niveau. Voici une liste de quelques en-têtes déclarant des types et fonctions de la bibliothèque standard :

- <assert.h> : pour un diagnostic de conception lors de l'exécution (assert)
- <ctype.h> : tests et classification des caractères (isalnum, tolower)
- <errno.h> : gestion minimale des erreurs (déclaration de la variable errno)
- <math.h> : fonctions mathématiques de base (sqrt, cos) ; nombreux ajouts en C99
- <signal.h> : gestion des signaux (signal et raise)
- <stddef.h> : définitions générales (déclaration de la constante NULL)
- <stdio.h> : pour les entrées/sorties de base (printf, scanf)
- <stdlib.h> : fonctions générales (malloc, rand)
- <string.h> : manipulation des chaînes de caractères (strcmp, strlen)
- <time.h> : manipulation du temps (time, ctime)

La bibliothèque standard normalisée n'offre aucun support de l'interface graphique, du réseau, des entrées/sorties sur port série ou parallèle, des systèmes temps réel, des processus, ou encore de la gestion avancée des erreurs (comme avec des exceptions structurées). Cela pourrait restreindre d'autant la portabilité pratique des programmes qui ont besoin de faire appel à certaines de ces fonctionnalités, sans l'existence de très nombreuses bibliothèques portables et palliant ce manque ; dans le monde UNIX, ce besoin a aussi fait émerger une autre norme, POSIX.1.

### Les bibliothèques externes

Le langage C étant un des langages les plus utilisés en programmation, de nombreuses bibliothèques ont été créées pour être utilisées avec le C. Fréquemment, lors de l'invention d'un format de données, une bibliothèque ou un logiciel de référence en C existe pour manipuler le format. C'est le cas pour libjpeg, libpng, Expat, les décodeurs de référence MPEG, libsocket, etc.

## Exemples

Voici quelques exemples présentant très succinctement quelques propriétés du C. Pour plus d'information, voir le WikiLivres "Programmation C".

### Allocation mémoire

La structure int\_list représente un élément d'une liste chaînée, contenant des données de type int. Les deux fonctions qui suivent (insert\_next et remove\_next) servent à ajouter et supprimer un élément de la liste.

```
/* La gestion de la mémoire n'est pas intégrée au langage  
mais assurée par des fonctions de la bibliothèque standard. */
```

```

#include <stdlib.h>

struct int_list {
    struct int_list *next; /* pointeur sur l'élément suivant */
    int value;             /* valeur de l'élément */
};

/*
 * Ajouter un élément à la suite d'un autre.
 * node : élément après lequel ajouter le nouveau
 * value : valeur de l'élément à ajouter
 * Retourne : adresse de l'élément ajouté, ou NULL en cas d'erreur.
 */
struct int_list *insert_next(struct int_list *node, int value) {
    /* Allocation de la mémoire pour un nouvel élément. */
    struct int_list *const new_next = malloc(sizeof *new_next);

    /* Si l'allocation a réussi, alors insérer new_next entre node
    et node->next. */
    if (new_next) {
        new_next->next = node->next;
        node->next = new_next;
        new_next->value = value;
    }

    return new_next;
}

/*
 * Supprimer l'élément suivant un autre.
 * node : élément dont le suivant est supprimé
 * Attention : comportement indéterminé s'il n'y pas d'élément suivant !
 */
void remove_next(struct int_list *node) {
    struct int_list *const node_to_remove = node->next;

    /* Retire l'élément suivant de la liste. */
    node->next = node->next->next;
    /* Libère la mémoire occupée par l'élément suivant. */
    free(node_to_remove);
}

```

Dans cet exemple, les deux fonctions essentielles sont `malloc` et `free`. La première sert à allouer de la mémoire, le paramètre qu'elle reçoit est le nombre de bytes que l'on désire allouer et elle retourne l'adresse du premier byte qui a été alloué, sinon elle retourne `NULL`. `free` sert à libérer la mémoire qui a été allouée par `malloc`.

## Références

- (en) « *The Development of the C Language* » (<http://cm.bell-labs.com/who/dmr/chist.html>) Dennis M. Ritchie.
- Dennis M. Ritchie et Brian W. Kernighan, *Le langage C*, Paris, Masson, 1986 [détail des éditions] (ISBN 2225800685, lire en ligne ([https://openlibrary.org/books/OL21056987M/Le\\_langage\\_C](https://openlibrary.org/books/OL21056987M/Le_langage_C))), p. 260,261
- (en) Samuel P. Harbison (ill. Guy L. Steele, Jr), *C, a reference manual* Upper Saddle River, N.J, Prentice-Hall, 2002, 533 p. (ISBN 978-0-130-89592-9 et 978-0-131-22560-2, OCLC 49820992 (<http://worldcat.org/oclc/49820992&lang=fr>)), p. 4
- (en), Thomas Wolf, *The New ISO Standard for C (C9X)*, 2000
- « ISO/IEC 9899:2011 - Technologies de l'information -- Langages de programmation -- C » ([http://www.iso.org/iso/fr/iso\\_catalogue/catalogue\\_tc/catalogue\\_detail.htm?csnumber=57853](http://www.iso.org/iso/fr/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853)) sur ISO (consulté le 1<sup>er</sup> février 2017)
- Ces particularités se retrouvent dans d'autres langages compilés tels que Fortran ou Ada (langage).
- Brian Kernighan et Dennis Ritchie (trad. Thierry Buffenoir), *Le langage C* [« The C Programming Language »], Paris, Masson, 1983, 1<sup>re</sup> éd., 218 p. [détail des éditions] (ISBN 2-225-80068-5), p. 4
- ISO 9899-2011, section 6.7.6.3, paragraphe 14
- ISO 9899-2011, section 6.11.6 : «*The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature*»
- MISRA C 2004, règle 16.5, p. 62
- (en) Brian Kernighan et Dennis Ritchie, *The C Programming Language* Prentice Hall, 1988, 2<sup>e</sup> éd., 272 p. [détail des éditions] (ISBN 0-13-110362-8), p. 106.
- « ISO/CEI 9899:TC3, Section 6.4.1: Keywords » (<http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf>) Organisation internationale de normalisation JTC1/SC22/WG14, 7 septembre 2007
- « ISO/CEI 9899:201x, Section 6.4.1: Keywords » (<http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf>) International Organization for Standardization JTC1/SC22/WG14, 12 avril 2011
- « Sizes of integer types », *ISO-IEC 9899*, 5.2.4.2.1





15. ISO/IEC 9899:1999 §3.4
16. ISO/IEC 9899:1999 §J.3 et §J.4
17. ISO/IEC 9899:1999 §6.2.6.2 paragraphe 2
18. **(en)** C Implementation-Defined Behavior: Integers implementation(<https://gcc.gnu.org/onlinedocs/gcc/Integers-implementation.html#Integers-implementation>)
19. ISO/IEC 9899:1999 §6.5.7 paragraphe 5
20. ISO/IEC 9899:1999 §J.1
21. ISO/IEC 9899:1999 §J.2
22. **(en)** [comp.lang.c FAQ list · Question 3.3](http://c-faq.com/expr/ieqplusplus.html)(<http://c-faq.com/expr/ieqplusplus.html>)
23. Plus précisément, il est autorisé d'accéder à un tableau dans ses limites ou un élément au-delà, pour faciliter les vérifications de débordement, mais pas plus loin.
24. **(en)** [What Every C Programmer Should Know About Undefined Behavior #1](http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html)(<http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>)
25. **(en)** [A Guide to Undefined Behavior in C and C++, Part 1](http://blog.regehr.org/archives/213)(<http://blog.regehr.org/archives/213>)
26. exemple issu de **(en)** [Linux Weekly - GCC and pointer overflows](https://lwn.net/Articles/278137/)(<https://lwn.net/Articles/278137/>)
27. **(en)** [Vulnerability Note VU#162289 - C compilers may silently discard some wraparound checks](https://www.kb.cert.org/vuls/id/162289)(<https://www.kb.cert.org/vuls/id/162289>)

## Voir aussi

### Articles connexes

- [Bibliothèque standard du C](#)
- [Alignement de données](#)
- [International Obfuscated C Code Contest](#)

Sur les autres projets Wikimedia :

-  [La programmation en C sur Wikiversity](#)
-  [Le langage C, sur Wikibooks](#)

### Quelques programmes célèbres écrits en C

- [UNIX](#)
- La suite de compilateurs [GNU Compiler Collection](#) (GCC)
- Le [noyau Linux](#)
- Le noyau de [Microsoft Windows](#)
- [GNOME](#)

## Bibliographie

- **(en)** [Brian Kernighan et Dennis Ritchie, \*The C Programming Language\*](#)[détail des éditions]
- [The international standardization working group for the programming language C](#)
  - **(en)** [ISO/CEI 9899:TC2 WG14/N1124](#) « Committee Draft », 6 mai 2005[lire en ligne]  
Le dernier *brouillon* de la norme ISO C99 incluant le *Technical Corrigendum 2*
  - **(en)** [ISO/CEI 9899:TC3 WG14/N1256](#) « Committee Draft », 7 septembre 2007[lire en ligne]  
Le dernier *brouillon* de la norme ISO C99 incluant le *Technical Corrigendum 3*
- **(en)** [Ivor Horton, \*Beginning C : from novice to professional\*](#)/Berkeley, CA New York, Apress Distributed to the Book trade in the U.S. by Springer-Verlag, 2006 (ISBN 978-1-590-59735-4 et 978-1-430-20243-1, OCLC 318290844, présentation en ligne)
- [Jean-Michel Léry, \*Le langage C\*](#), Paris, Pearson Education France, coll. « Synthex., Informatique », 2005 (1<sup>re</sup> éd. 2002), 303 p. (ISBN 978-2-744-07086-0, OCLC 77036023, notice BnF n° FRBNF39933122)
- [Achille Braquelaire, \*Méthodologie de la programmation en C\*](#) Dunod, 2005 (ISBN 978-2-10-049018-9)
- [Claude Delannoy, \*Programmer en langage C : cours et exercices corrigés\*](#) Paris, Ed. Eyrolles, coll. « Collection noire », 2002, 267 p. (ISBN 978-2-212-11072-2, OCLC 50208529), 11<sup>e</sup> tirage
- [Mathieu Nebra, \*Apprenez à programmer en C : enfin un livre pour les débutants\*](#) Paris, Simple IT, coll. « Le Livre du zéro », 2012, 2<sup>e</sup> éd. (1<sup>re</sup> éd. 2009), 526 p. (ISBN 979-1-090-08500-8, OCLC 793485436)

- Éric Berthomier et Daniel Schang, *Le C en 20 heures*, Framasoft, coll. « Framabook » (n° 6), 2013, 3<sup>e</sup> éd. (1<sup>re</sup> éd. 2010), 196 p. ([ISBN 978-2-9539187-7-9](#), [présentation en ligne](#), [lire en ligne](#))

---

Ce document provient de «[https://fr.wikipedia.org/w/index.php?title=C\\_\(langage\)&oldid=146718828](https://fr.wikipedia.org/w/index.php?title=C_(langage)&oldid=146718828)».

**La dernière modification de cette page a été faite le 23 mars 2018 à 15:39.**

Droit d'auteur : les textes sont disponibles sous licence Creative Commons attribution, partage dans les mêmes conditions ; d'autres conditions peuvent s'appliquer. Voyez les [conditions d'utilisation](#) pour plus de détails, ainsi que les [crédits graphiques](#). En cas de réutilisation des textes de cette page, voyez [comment citer les auteurs et mentionner la licence](#).

Wikipedia® est une marque déposée de la [Wikimedia Foundation, Inc.](#), organisation de bienfaisance régie par le paragraphe [501\(c\)\(3\)](#) du code fiscal des États-Unis.